RESEARCH NOTE 80-18

LEVEL II

(2)

MODERN PROGRAMMING PRACTICES: IMPLICATIONS FOR HUMAN FACTORS RESEARCH

Ruven Brooks and Michael G. Samet
Perceptronics, Inc.

AD A096874

HUMAN FACTORS TECHNICAL AREA

U. S. Army
Research Institute for the Behavioral and Social Sciences

JULY 1979

U. S. ARMY RESEARCH INSTITUTE FOR THE BEHAVIORAL AND SOCIAL SCIENCES

A Field Operating Agency under the Jurisdiction of the Deputy Chief of Staff for Personnel

JOSEPH ZEIDNER
Technical Director

FRANKLIN A. HART
Colonel, US Army
Commander

NOTICES

DISTRIBUTION: Primary distribution of this report has been made by ARI. Please address correspondence concerning distribution of reports to: U. S. Army Research Institute for the Behavioral and Social Sciences, ATTN: PERI-TP, 5001 Eisenhower Avenue, Alexandria, Virginia 22333.

FINAL DISPOSITION: This report may be destroyed when it is no longer needed. Please do not return it to the U. S. Army Research Institute for the Behavioral and Social Sciences.

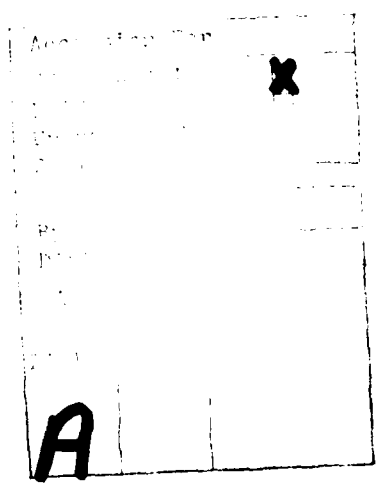
NOTE: The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

RESEARCH NOTE 80-18

MODERN PROGRAMMING PRACTICES: IMPLICATIONS FOR HUMAN FACTORS RESEARCH

Ruven Brooks and Michael G. Samet
Perceptronics, Inc.

HUMAN FACTORS TECHNICAL AREA



U. S. Army
Research Institute for the Behavioral and Social Sciences

JULY 1979

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Research Note 80-18	2. GOVT ACCESSION NO. AD-A096	3. RECIPIENT'S CATALOG NUMBER 874
4. TITLE (and Subtitle) Modern Programming Practices: Implications for Human Factors Research		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER PFTR-1045-79-7
7. AUTHOR(s) Ruven Brooks and Michael G. Samet		8. CONTRACT OR GRANT NUMBER(s) DAHC19-77-C-0012
9. PERFORMING ORGANIZATION NAME AND ADDRESS U.S. Army Research Institute for the Behavioral and Social Sciences (PERI-OS) 5001 Eisenhower Ave. Alexandria VA 22333		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 2Q762717A765
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE July 1979 13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Jean A. Nichols of ARI provided valuable comments on the draft manuscript.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Software	Programming Conventions	Software Management
Human Factors	Programming Practice	Software Testing
Program Design	Software Design	Software Tools
Program Specification	Software Evaluation	Structured Programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Future research directions on human factors in software must be sensitive to issues arising out of evolving software practice. Based on a historical distinction, these practices can be divided into two groups, conventional and modern. A series of reports prepared by six large software contractors on the impact of modern programming methods provides a useful source for evaluating the effectiveness of various practices. This paper critically analyzes selected contents of these reports and summarizes their conclusions

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (Continued)

as to the practical impact of modern programming practices. In light of the results of this analysis, a set of future research directions for work on human factors in software is then suggested.

TABLE OF CONTENTS

ABSTRACT

Future research directions on human factors in software must be sensitive to issues arising out of evolving software practice. Based on a historical distinction, these practices can be divided into two groups, conventional and modern. A series of reports prepared by six large software contractors on the impact of modern programming methods provides a useful source for evaluating the effectiveness of various practices. This paper critically analyzes selected contents of these reports and summarizes their conclusions as to the practical impact of modern programming practices. In light of the results of this analysis, a set of future research directions for work on human factors in software is then suggested.

1. INTRODUCTION

The past decade has seen the emergence of a crisis in the timely and cost-effective production of software (Boehm, 1973; Brooks, 1975). In response to this crisis, a number of new software practices and techniques, bearing such names as structured programming (Dahl, Dijkstra, and Hoare, 1972), modular programming (Parnas, 1972), and top-down design (Stevens, Meyers, and Constantine, 1974) have been developed and advocated as useful in combating this crisis. At the same time, and perhaps, with the same motivation, there has been an increasing interest in what might be called human factors in software development (e.g., see Atwood, Ramsey, Hooper, and Kullas, 1979; Shneiderman, 1979; Weinberg, 1971). This area involves experimental study of human behavior in software production. For example, research topics include errors in programming (Youngs, 1974), the merits of various programming language constructs (Sime, Arblaster, and Green, 1977; Sime, Green, and Guest, 1973, 1977), debugging (Gould and Drongowski, 1974), and the cognitive behavior in writing programs (Sussman, 1976; Brooks, 1977).

If this human factors research area is to continue to grow and be of influence, it is important that attention be paid to issues that emerge in the development of improved software practices (e.g., Ramsey, Atwood, and Campbell, 1979). The role of these human factors issues is, in turn, dependent on which of the emerging practices are found most effective in software production and maintenance. The effectiveness of various practices is, of course, still a very open question and the outcome of studies done in controlled experimental settings does not necessarily predict the outcome of use of the practices in applied situations. Furthermore, very few controlled experimental evaluations of programming practices and techniques have been conducted.

Recently, a series of reports has become available which describes how these new approaches actually work in practice. Prepared for the Air Force, these reports describe the impact of what is referred to as "modern programming practice" (MPP) on projects done by six major defense software contractors: Computer Science Corporation (Donahoo, Carter, Hurt, and Farquhar, 1977), System Development Corporation (Perry and Willmorth, 1977), TRW (Brown, 1977), Sperry-Univac (Branning, Willson, Schenzer, and Erickson, 1977), Boeing Computer Services (Black, Katz, Gray, and Curnow, 1977), and Martin-Marietta Corporation (Prentiss, 1977). Some of the reports describe projects that involve among the largest software systems ever built; thus, the content of these reports are a useful basis for determining future directions in research on human factors and software. This report critically reviews the results of MPP experience from the perspective of human factors, to draw conclusions concerning the utility of selected MPP and to identify potential research issues.

2. A VIEWPOINT ON MODERN PROGRAMMING PRACTICE

2.1 Era I - Machine Capabilities As A Primary Constraint

Since there is still considerable discussion as to which practices should be considered as "modern" and since each of the reports describes the meaning of the phrase somewhat differently, it is a necessary preliminary to define what constitutes a modern programming practice. One way to make this definition is in terms of two successive eras of software practice. The first era can be viewed as beginning with the first digital systems and terminating, about 1965, with the introduction of hardware based on integrated circuits and the establishment of the first university computer science departments. Computing costs in this era were overwhelmingly dominated by the costs of hardware (Boehm, 1973), and the biggest limiting factor on the development of new computing applications was the availability of machine resources.

2.2 Era II - Software As A Critical Constraint

The second era, which extends to the present time, is, in contrast, characterized by software as the most critical and most costly factor in computing applications (Boehm, 1973). As machine resources have become readily available, the size of programs has also grown, and most sizeable programs now require the efforts of many programmers. Because of the complexity of the systems, a substantial level of computer science expertise is required for system construction and modification. This shift in the nature of software has also produced a shift in the type of problems encountered in computing projects. Formerly, the strongest constraints on project completion were those of the available computing time with which to accomplish the project. Now, since development techniques have not kept pace with the increasing complexity of software requirements, the constraints are more likely to be those of the labor available. As a

consequence, cost and schedule overruns are frequent. Additionally, since the system builders are no longer the domain experts, there is an increasing problem to ensure that the software performs the right set of functions desired by the customer.

2.3 Modern Versus Conventional Programming Practice

With the increasing size and complexity of modern software projects, several problem areas have developed which are unique to this second era of computing. These problems include how to coordinate development of software across multiple programmers, how to ensure maintainability of a software project throughout its life cycle, and how to adequately analyze and translate user requirements into a finished software product. Practices intended primarily to attack second-era problems constitute modern programming practice. They stand in contrast to practices designed primarily to enhance the availability and accessibility of machine resources, which will be referred to here as conventional programming practices.

2.3.1 Two Examples of Modern Programming Practice. A modern programming practice (MPP), intended to help ensure that the delivered program meets the customer's needs, is the use of a requirements baseline. This is a document, written before the start of programming, that specifies the functions to be performed by the program. It is reviewed and verified by the customer. This practice stands in contrast to starting programming with only an informal idea of what the customer wants and then reprogramming the system as the customer's real needs are revealed.

Another modern practice to ensure coordination among programmers is the use of formal design documents that completely specify the internal structure and organization of a program. Without such documents, it is difficult or impossible to ensure that parts of the program written by different programmers will interface properly, and considerable time and effort will be wasted in rewriting the code so that they do fit.

2.3.2 Two Examples of Conventional Programming Practice. In contrast to the two MPPs described above, it is useful to view two conventional practices that are relatively recent in origin but which still have as their goal the enhancement of machine resource availability. The first of these is the use of symbolic debuggers. A symbolic debugger is a tool for debugging compiled code that allows the user to refer to memory locations by the variable names from the source language program. While such a tool can be an extremely powerful programming aid, it still must be viewed primarily as a substitute for machine resources, in this case, for enough machine power to use an interpreter instead of a compiler.

A second example of conventional practice is the use of hardware emulation in which one machine can be programmed to execute the instruction set of another. This means that a single machine can be used as a vehicle for software development for a number of target machines. While this practice can be of substantial benefit, particularly where software development is done in parallel with hardware work, it still has as its main purposes increasing hardware availability. Of course, if the target machine is readily available, emulation is unnecessary.

A concluding remark about the distinction between conventional and modern programming practice is that a single practice may fall into either category, depending on the purpose for which it is used. An example is the use of higher-order languages (HOLs). If they are used to increase machine accessibility, so that the individual programmer can write more code faster, then the use of HOLs is a conventional practice. If they are used to improve the readability of the code by other programmers, then they fall into the modern practice category.

2.4 Classification of MPP

In addition to serving the goals described earlier, MPP can be classified according to the way in which they affect software production as being either: technical, managerial, or informational. Technical practices are those which affect the tools used by the individual programmer to perform work and that might be used if the programmer were the only person working on the project; the use of languages with control structures for structured programming falls into this category. Managerial practices are those that affect the allocation or resources and responsibility; the use of a chief programmer team is an example of a managerial practice. Informational practices are intended to transmit information among the members of a programming group; an example of an informational practice is the holding of design reviews. Informational practices may also have impact on project management because they play a role in informing managers how well a project is proceeding, but they have no direct role in assigning work or responsibility. Any single practice can, of course, have major effects in more than one category.

The following section describes several representative MPP, selected mainly as a result of their commonality across the reports. Because of similarities between certain MPPs (and because they may be referred to by different names in different reports), the practices are presented in related sets: namely, scheduling practices, design practices, programming practices, and validation practices. Along with each practice, its classification by effect on software production is given (i.e., either technical, managerial, informational, or some combination of these). A categorized list of the selected MPP and their effects is provided in Table 2-1.

TABLE 2-1

CLASSIFICATION OF SELECTED MODERN PROGRAMMING PRACTICES (MPP) WITH REGARD TO AREAS OF PRIMARY IMPACT

Practice	Technical	Managerial	Informational
Group A - Implementation Scheduling Practices			
(1) Incremental Implementation		'	'
(2) Build		'	'
(3) Configuration Management			'
(4) Stub	'		
(5) Thread			'
Group B - Design Practices			
(6) Top-Down Design	'		'
(7) HIPO Diagram	'		'
(8) Program Design Language (PDL)	'		
(9) Requirements Baseline		'	'
Group C - Programming Practices			
(10) Programming Conventions	'		'
(11) Structured Programming	'		
(12) Modular Programming	'		'
Group D - Validation Practices			
(13) Independent Testing		'	
(14) Test Scenarios and Test Specifications			'
(15) Requirements, Design, and Code Reviews			'
Group E - Management Practices			
(16) Program Library	'		'
(17) Chief-Programmer Team		'	

2.5 Some Representative MPP

Group A - Implementation Scheduling Practices

(1) Incremental Implementation

Classification: managerial, informational

Incremental implementation refers to the practice of constructing a system as a series of successively more complete partial systems, termed builds. Each build possesses a subset of the functions of the complete system. Two primary benefits are claimed for this technique. From a managerial viewpoint, it has the benefit that, at any given point in time, a known portion of the entire project has been completed. Not only is this beneficial for internal management visibility, but it can also be used to demonstrate project status to the customer. From an informational viewpoint, this implementation method has the advantage that it eliminates much of the need for writing separate test drivers for each module, since the partial system can be used as the test environment for subsequent modules.

(2) Build

Classification: managerial, informational

A build is a completion stage in an incremental implementation of a system. It consists of sufficient parts of the eventual system to perform some subset of the functions of the entire system when it is complete. Unimplemented parts of a system are represented by stubs that have the same calling parameters or core requirements as the eventual code.

but that do no computation. A new build in constructed from a prior one by replacement of some of the stubs with working code. As an example, consider a system to generate five different kinds of managerial reports. The first build on the system might be able to generate two of these reports; the second build might add two more of the reports, and the final build would have all of the capabilities. A defining characteristic of builds is that, from the customer's standpoint, each build is a useable system. The claimed benefits of the use of builds are that they lead to earlier discovery of errors or problems in the system and that they provide useful managerial information on the progress in system construction.

(3) Configuration Management

Classification: managerial, informational

If a large software system is being constructured in an incremental fashion, modifications must often be made to the partially assembled system, usually as a result of discovering deficiencies or errors. This remedial work must usually be done in parallel with further additions to the system. If the remedial work is done in an undisciplined fashion, situations may arise in which those doing further construction on the system are unable to tell whether problems are due to the new software that are adding or to the remedial work. Configuration management refers to a systematic mechanism for deciding when modifications should be made and for informing those affected.

A frequently adopted technique is to implement the system in a succession of discrete steps, or builds. (See incremental Implementation.) The majority of remedial changes

are made at the time a new build is assembled. An organization referred to as a change control board is given the responsibility of reviewing proposed modifications, assessing their impact on the system, and determining if they should be made immediately or deferred to the next build. The claimed benefit of configuration management is avoidance of the type of software instability described earlier.

(4) Stub

Classification: technical

A stub is a piece of code in a build used as a place holder for a section of the system that has not yet been written. It is usually designed to have the same general form as the code that will eventually replace it. For example, if the stub is for a subroutine, it will be defined to have the same type and number of arguments as the eventual subroutine, and it will return a constant value of appropriate type and range. If desirable, it may also print a message saying that execution has taken place. If size and time are important constraints on the system, the stub may also be constructed to occupy the same storage and to take the same amount of execution time as the eventual code.

(5) Thread

Classification: informational

A thread is a specification of system function that consists of the inputs needed for the function, the processing required by it, and the output produced from it. A full system consists of multiple, intertwining threads. The identification

of threads is often useful in selecting which systems component to include in a build. As an example, if a system can be broken down into 18 distinct threads, 6 of them may be selected to be implemented in the first build.

Group B - Design Practices

(6) Top-Down Design

Classification: technical, informational

Top-down design refers to a methodology for system design by starting with a global, high-level system and refining it in a series of successively more specific levels of detail. All parts of the system are refined to one level before work begins on the next level. At each level, the refinement process begins with those modules most central to the computation and furthest removed from the input and output.

This procedure is distinguished from bottom-up design, which attempts to design a few, common, low-level modules at a fine level of detail and then builds the rest of the design around these modules. Designs also can be created unsystematically with order of design decisions dependent on the choice of the designers. The claimed benefit of top-down design is that it avoids premature decisions on unimportant aspects of a system that may unnecessarily restrict decisions about more critical aspects of the system.

(7) HIPO Diagram

Classification: technical, informational

HIPO stands for hierarchy plus input/process/output. A HIPO diagram shows for each process the input used by that process, the functions performed by it, and the output produced by it. The charts are organized hierarchically, so that part of a process at one level may be represented by a complete HIPO diagram at another level. HIPO diagrams are claimed to be an improvement over flowcharts in that they emphasize the data and the function performed rather than the processing steps. In addition, they are better than flowcharts for showing modular decomposition.

(8) Program Design Language (structured narrative)

Classification: technical

Program design languages (PDLs) are notational systems or languages for describing the data and control structure and general organization of a computer program. They differ from conventional programming languages in that the semantics (and, sometimes, the syntax as well) are not formally specified, but are selected by the programmer to convey particular meanings. An example of an acceptable statement in some PDLs is "WHILE there are more cases to process, DO get next case and add up total." A given program may be described by a number of successive layers of PDL, each describing the program at a finer level of detail.

In comparison with flowcharts or HIPO diagrams, PDLs are claimed to be easier for the programmer to use since they

do not require the use of special symbols, allow the use of descriptive names of unlimited length, and are compatible with conventional text editors and word processing equipment. In addition, PDLs may be easier to translate into the eventual programming language. Finally, PDLs are claimed to be more readable by other programmers and managers (see Ramsey, Atwood, and Van Doren, 1978).

(9) Requirements Baseline

Classification: managerial, informational

Requirements baseline refers to a formal description in functional terms, as independent as possible from a particular implementation, of the tasks that a program or system is supposed to perform. Ideally, this baseline remains constant throughout the system development process and serves as a standard against which the final product can be assessed. This practice is designed to prevent the situation in which the requirements for the system are so hazy at the start of construction that the functions performed by the system must be determined in the course of the construction process. Not only may this lead to frequent revisions of the system design, but it also frequently leads to customer dissatisfaction with the delivered system.

Group C - Programming Conventions

(10) Programming Conventions

Classification: technical, informational

Programming conventions are a set of standards of the way code is to be written and then followed by all of the programmers on a project. The aim of these standards is to

improve the readability and comprehensibility of the code. Often included are rules for the formation of identifier names, rules for indenting and spacing of code, and standards for commenting the program. Adherence to these standards is often aided by software tools, such as formatters.

(11) Structured Programming

Classification: technical

Structured programming is used here in the restricted sense of programming techniques that result in programs whose associated flowgraphs possess the property of reducibility.* This result may be achieved by the use of certain sets of control structure primitives, such as IF-THEN-ELSE, DO-WHILE, CONTINUE, and BREAK. These sets possess the property that any programs written in them have reducible flowgraphs. Alternately, the same effect may be achieved by the use of certain coding conventions in languages with other control structures. Claimed benefits of using the structured programming technique are that the resultant programs are easier to read and understand and that they are easier to decompose functionally, which has important benefits for program refinement and optimization.

(12) Modular Programming

Classification: technical, informational

In the modular programming methodology, a system is defined as a collection of pieces or modules, such that the internal

*A flowgraph is reducible if the edges can be partitioned into two disjoint groups of forward and back edges such that the forward edges form a connected, acyclic graph, and the back edges consist only of edges whose heads dominate their tails.

organization or structure of each module is as independent as possible from that of other modules. This carries the implication that each module performs a single, simple function and that each module passes the minimum necessary information to other modules (information hiding). A number of techniques are available to ensure this modularity. One is to require that each module return only to the calling module. A second is to use access functions to protect the integrity of common data structures.

Claimed benefits of this technique are, first, that good modularization makes the division of the programming task among individuals easier, since the independence of the modules reduces the need for coordination among individuals. Second, it reduces errors because the programmer need only concern himself with writing code to perform a single, simple function. Third, if errors do occur, good modularization is claimed to make isolation of errors easier. Finally, if the program needs to be modified, the modification can often be confined only to the function being changed.

Group D - Validation Practices

(13) Independent Testing

Classification: managerial

In some projects, the testing of a system is carried out by a different part of the organization than that which originally provided the code. This independent testing is carried out after system integration and is aimed at determining whether there are any errors in the system and whether the system meets the requirements. The claimed benefits for this approach are greater objectivity and rigor in testing.

(14) Test Scenarios and Test Specifications
Classification: informational

Test scenarios and test specifications are alternate ways of referring to the practice of preparing formal descriptions of the testing that should be performed on a build or on a complete system. These techniques stand in contrast to informal test procedures that are left to the discretion of the system implementors. Given such formal descriptions, the actual testing may be carried out either by the implementation group or by an independent test organization. The claimed benefits of this approach are the same as those for independent testing, namely greater objectivity and rigor in testing.

(15) Requirements, Design, and Code Reviews
Classification: informational

Requirements, design, and code reviews are terms that refer to the verification of the completeness and accuracy of a stage in system development by holding formal reviews of the quality of the system. The mechanisms used include oral presentations, inspection of documents, and informal simulations or walkthroughs. The requirements review is usually conducted by the customer together with the organization building the system. Design reviews often involve the whole development staff, while code reviews usually involve only one or two other programmers. The intent of such reviews is to uncover errors and inconsistencies as early as possible, before they can impact on further system construction.

Group E - Miscellaneous

(16) Program Library

Classification: technical, managerial, informational
Related concept: team or project librarian

A program library is any organized system for ensuring access by different personnel to common collections of software. The software can include individual source and object modules, data, macro and parameter definitions; partial or preliminary versions of entire systems; commonly used simulations and test generations; and error or exception reports and documentation.

A number of goals are claimed to be advanced by this technique. These include:

(1) Standardization of module interfaces through common data and parameter definitions.
(2) Providing a stable debugging environment by controlling access to partially built systems.
(3) Increased management visibility of project status by the use of status information associated with library entries.
(4) Ensuring adherence to coding conventions by making adherence a precondition to library entry.
(5) Ensuring updating of documentation by requiring simultaneous modification of a program and its documentation.
(6) Protecting system integrity by keeping modification history and backup versions.

Program libraries can be implemented mechanically by controlling access to card decks or they can be accomplished through software such as loaders and special library control programs.

(17) Chief Programmer Team

Classification: managerial

This term refers to a particular organization of programming groups in which a single individual, the chief programmer, assumes complete technical and managerial responsibility for a programming project throughout the design and implementation phases of the project. Besides the chief programmer, the team consists of a backup programmer, who assists and backs up the chief programmer, and a project secretary/librarian. In addition to these core members, other team members may be added as needed for various durations. Typical long-term additions might include additional support programmers, analysts and administrative assistants. Short-term additions might include specialists in finance or contracts and programmers specializing in one area, such as file I/O.

The claimed benefit of this organization is that, for medium scale projects (less than 100,000 lines of code), placing the technical responsibility in the hands of a single highly competent individual eliminates the problems of coordination that result when this responsibility is fragmented and spread among a group of individuals. Additionally, placing of managerial responsibility in the same hands as the technical direction insures harmony between technical and managerial goals.

2.6 Summary

A historically based distinction has been made between conventional programming practices, intended to enhance machine availability and accessibility, and modern programming practices (MPPs) that have as goals: (a) insuring that the delivered software meets the user's needs; (b) coordinating among many programmers working on the same project; and (c) increasing the ease with which software can be modified and maintained. Within the general class of MPP, a further distinction has been made between technical, managerial, and informational practices.

3. THE IMPACT OF MODERN PROGRAMMING PRACTICES

3.1 The Individual Experiences

The reports from the six contractors represent a wide range of programming experience. They vary in tasks, size, effort expended, as well as in machines and languages used. In terms of tasks performed, they included database management systems, scientific data analysis systems, and on-board real-time computer systems. The smallest project reported on contained less than 3,000 lines of code and took just over half a person-year to complete; the largest project involved about a million lines of code and 600-person years of effort. Machines used ranged from mini-computers to the largest commercially available processors; and languages included FORTRAN, ALGOL, CMS-2, JOVIAL, and a variety of assembly languages. In addition, the reports describe the utilization of different analysis methods for arriving at conclusions, including interviews with project staff, application of the delphi technique, estimation of comparative program sizes, and examination of project documents. Finally, across the projects many different modern programming practices (MPPs) were employed and reported on. Selected programming experiences of each contractor are briefly described and analyzed in this chapter; the specific projects covered are cataloged in Table 3-1, which also summarizes project characteristics.

3.1.1 The System Development Corporation Experience. The System Development Corporation (SDC) report (Perry and Willmorth, 1977) and recommendations are based on experience from two projects, COBRA DANE, a high-technology space surveillance system, and the Air Force Satellite Control Facility's COMPOOL - Sensitive System, a software tool, similar to a database management system, used to define and control data structures shared by a number of programs. The COBRA DANE project involved approximately 50 technical staff and produced about 40,000

TABLE 3-1

SUMMARY OF CONTRACTOR PROJECTS

CONTRACTOR PROJECT	SIZE[1]	EFFORT[2]	LANGUAGE	ANALYSIS[3]	PRACTICES USED
Systems Development Corporation[4]					
Space Surveillance	40,000	60	FORTRAN	I D	1,2,3,10, 12,15,17
Database Management	??	??	JOVIAL	I D	See note 4.
TRW					
Satellite Tracking	1,000,000	600(?)	FORTRAN PCP	D	1,2,3,5 6,9,10,11 12,13,14, 15,16,17
Sperry-Univac					
Navigational Program	90,000	52.5	CMS-2Y	A	3,3,14
Simulation Program	500,000	246.7	CMS-2Y	A	3,14
Command/Control Operating System	36,600	292.2	CMS-2Y	A	1,2,3,4, 6,12,14
Real-Time Process Control Program	13,150	188.2	CMS-2Y	A	1,2,3,4, 6,8,12,14
Boeing Computer Services					
Configuration Accountability System	13,800 3,800	6.9	ALGOL Assembler	E	1,2,6,8,9,16
Estimator Modeling Language	480,000	92.6	COBOL	E	1,2,3,9, 8,11,15,16
Management Information	40,000	3.6	FORTRAN	E	1,2,6,11,12 8,9,11,15,16
Resource Accounting	396,000	42.9	COBOL	E	1,2,3,6,9, 11,15,16
Mailing Lists	2,890	0.6	FORTRAN	E	1,2,6,9,11,15
Computer Science Corporation					
Missile Control	310,000 words	33(?)	CMS-2	I	1,2,3,4,5,6,9, 10,12,13,14, 15,16
Data Acquistion	94,000 words	69(?)	CMS-2	I	1,2,3,5,6, 9,10,11
Command/Control	250,000	90(?)	CMS-2	I	1,2,3,5,6,9, 10,11,12,13, 14,17
Martin-Marietta					
Viking Mission Planning (22 programs)	278,575 cards	148.6	FORTRAN + Assembler	A	1,2,9,15,17
Viking Lander Flight Program	400,000	175.0	Assembler Instructions	A	9,12,13,15
Viking Systems	133,000	50.0	Assembler	A	3,9,14

[1] Size indicates line of code unless otherwise specified.

[2] Effort denotes number of person-years.

[3] Analysis Techniques: I = interviews, D = Delphi technique, E = estimation of comparative program sizes, A = examination of project documents.

[4] The COMPOOL project was actually the development of a tool, rather than the use of a tool in a project, and it involved very little new code writing; hence, no use of PPP is known for it.

statements, mostly in FORTRAN, but with some assembly language subroutines. The COMPOOL project consisted mainly of revisions to existing programs and systems, so that no overall size figure is easily obtainable, but the magnitude of effort seemed similar to that of the COBRA DANE project.

The data collected to evaluate the practices used on the COBRA DANE study consisted of source program listings, design documentation, program library logs, minutes and records of the configuration control boards, interviews with documentation writers and test team members, and interviews with project members and managers. These data were not quantified or analyzed in any statistical or formal manner, so the conclusions of the report are based on the impressions and insights of the report's authors. The analysis of the COMPOOL system was oriented towards the impact of the COMPOOL system on the other projects that used it, not on the practices used in constructing the system itself. As in the COBRA DANE case, the analysis was qualitative and impressionistic.

COBRA DANE Project

The COBRA DANE project used five major MPPs: configuration management, programming conventions, program library, chief-programmer teams, and data structure management. The major findings with respect to each of these practices are summarized below.

Configuration Management. The term, configuration management or configuration control, as used in COBRA DANE included the holding of preliminary and critical design reviews, the specification of testing of builds, and monitoring changes and updates to builds. Normally, only the last function is included under configuration control, but all these functions were apparently grouped together under this heading since they were all under the control of the same managerial structure. This structure con-

sisted of two control boards, a joint configuration control board, which handled hardware changed and their interface to the software, and an internal configuration control board responsible for internal software design.

The grouping of all of these functions under this one organizational structure was, in turn, due to a failure to create a firm set of performance specifications for the system before work on it was begun -- the final performance specifications were, in fact, written after completion of the software! This led to constant revisions of the design, as well as of the code, which were all coordinated by the change control boards. Surprisingly, this problem did not appear to impact on system delivery, though the error rate in the delivered code was somewhat high. Whether this is because the design changes were, in fact, minor or because the change control mechanism overcame the problem is not clear from the report.

Programming Conventions. The programming conventions used consisted of an explicit and detailed set of commenting conventions, indenting and paragraphing conventions, and module and variable-naming conventions. Also included under this heading was the division of the system into modules, since this was partially under the control of the individual programmers and was indicated by the use of modularity conventions. No particular attempt was made to follow the conventions for structured programming in FORTRAN.

All the conventions were seen as having a net positive effect on programming effort, but no estimates were made on the magnitude of this effect. The authors of the report did seem to feel that the comments were particularly valuable, noting that maintenance programmers often inserted such commentary if it were missing or inadequate. They also noted that programmers felt that annotation of data items was more important than that of processing steps.

Program Library. An incremental implementation policy was followed in the construction of the system, and the main goal of the system library was to control changes to the builds. In addition to custody over the builds, the system library also contained a database on software errors and problems. Rather than having team librarians, a single system librarian maintained a single library for the whole project. After approval of any changes by a control board, this librarian added update decks supplied by the programmers to the system library, along with control information about the status of the program. The librarian was not, however, responsible for the accuracy of the programs. An aspect unique to this project was that the system librarian, not the programmers, was responsible for providing stubs for incomplete systems components.

The program library was considered to have met its main objectives of providing close management over the developing product and of improving programming efficiency and convenience. However, the use of a single library for all teams apparently was not entirely successful. In the latter stages of the project, changes to the system were made at a high rate, and each of the programming teams began to maintain its own version of each build to ensure a stable enough environment for completion of their part of the system.

Chief-Programmer Team. The COBRA DANE structure departed from a formal chief programmer team architecture in the use of a single librarian for the whole system rather than having separate team libraries. While the overall impact of chief programmer team organization was believed to have been extremely beneficial, two problems were noted. The first was a tendency for the chief programmers to give higher priority to technical than to administrative matters, with the consequence that unresolved inter-personal friction may have, on occasion, disrupted plans and reduced morale. A second problem was that, since there were multiple teams,

a need arose for coordination among chief programmers. The notion of chief architect was suggested in the project report, but the need for such an individual may be only a manifestation of the lack of clear performance specifications.

Data Structure Management. Because many different parts of the system used the same common data, a mechanism was needed to specify the organization of the data. COBRA DANE set up a database library, similar to the program library, which was administered by a database coordinator. The jobs performed by the coordinator were, among others, the naming of global database entries, giving initial values to system constants, checking adherence to naming conventions for global data, and providing listings of database contents. As with the program library, changes to the database had to go through change control procedures, and the database coordinator was not responsible for maintaining the correctness of programs in regard to these changes. While the database library was seen as having an essentially positive impact, it did not eliminate the problems of uncoordinated changes to the database. This was apparently because there was no mechanism for automatically updating programs that were affected by a database change, so that changes "tended to ripple through the system, impacting in unexpected places."

COMPCOL Project

The COMPOOL project involved a data management system designed to ease the interchange and common usage of data between programs. The system consisted of a number of components, such as a compiler that could access common data definitions, loaders sensitive to symbolic names, and managerial procedures for reviewing and controlling changes to the database. No actual data on the impact of the system is presented in the report; however, perceived benefits fall into three groups: work specification, improved resource utilization, and configuration management. The work simplification was seen as occurring because the number of data

declarations needed was reduced, because automated load and link operations were provided, and because a symbolic debugger was available. Improved resource utilization was seen as a result of improved storage sharing between programs and of the elimination of data redundancies. Finally, the standardization of data structures and interfaces and the centralization of data sources were held to aid overall system management.

3.1.2 The TRW Experience. The report from TRW (Brown, 1977) is based on the experience with the Systems Technology Program (STP), a descendent of a real-time processing system originally designed as part of the site defense system for the Minuteman system. At peak, the project employed more than 400 staff and the total size of the project exceeded 1,000,000 instructions. The system requires a dedicated CDC-7600 computer.

The report on the role of MPP in the STP project was based on data collected from two surveys of project personnel. The first survey concerned general MPP and was constructed using a modified Delphi technique in which preliminary surveys were used to select those MPP that were most likely to be important. A total of 11 MPP were included in the final survey. In the final form of the questionnaire, respondents were first asked to rank the impact (positive or negative) of each of the 11 MPP on each of 12 software project problems, such as schedule overrun or lack of conclusive testing. They were then asked to rank order the problems by importance and to rank the MPP by their overall impact on productivity. A total of 67 respondents were used, representing a cross section of STP personnel.

The second survey was intended to gather data on the impact of specific programming practices. A total of 54 raters, again, selected to be a cross section of project personnel, were asked to rate each of 18 programming practices as to its effect on each of 30 software characteristics. Examples of the practices are the use of a particular statement label format, the use of only labeled common, and the use of variable naming conventions. Examples of the characteristics are code auditability, testing thoroughness, and execution time.

A number of different analyses were carried out on the data from the first survey to explore the relationship between MPP and perceived problems. These analyses revealed that the most significant problems were cost overrun and inadequate satisfaction of real requirements. As might be expected from these problems, the MPP deemed of greatest value was requirements analysis and validation. Following closely were baselining of requirements specification and the completion of an entire preliminary design (before detailed design and coding). Of low value were enforced programming standards, independent testing, and software configuration management, consistent with low rankings of maintainability and poor documentation as problems.

An interpretation of these findings can be made on the basis of a situation not uncommon in projects of the magnitude of STP. During the five year life of the project, the goals of the project underwent several major changes, mainly as a result of congressional action. In such a situation, it may have been the case that the delivered software, no matter how bug-free, failed to perform the needed functions. If the differences were large enough so that extensive sections of code had to be scrapped and written anew, then it is expected that maintainability would receive a low rating as a problem. Practices such as independent testing and configuration management would, consequently, be seen as having lesser benefit.

The results of the second survey show that strong positive impact on software was shared by the use of four programming practices: detailed text description of the purpose, function, and interfaces of modules; macro flowcharts describing system organization and the organization of individual routines; preface commentary standards; and in-line commentary standards. These can all be described as giving information on code functioning that is not an integral part of the code itself. Practices such as naming conventions or modularity conventions, which could be considered integral to the code, received weaker positive ratings.

3.1.3 The Sperry-Univac Experience. The Sperry-Univac report (Branning, Wilson, Erickson, and Schaenzer, 1977) draws on the experience derived from the use of MPP on the development of four naval command and control systems. The systems were coded mostly in the CMS higher-level language with a range of assembly language usage that peaked at 34% for one project. For two of the projects, nearly all software development was done on a host machine for a smaller target machine. Project sizes ranged from 13,150 to 500,000 source lines of code, with manpower utilization ranging from 9.5 to 246.6 person-years. An interesting feature of three out of the four projects is that they were additions or upgrades to existing programs.

The methodology used to collect data on MPP impact was a survey of project personnel about the tools and techniques used and their perceived effectiveness. No formal analysis was carried out on the effectiveness estimations; however, in regard to one group of practices, a formal effectiveness measure was used. This measure was obtained by making a performance rating for each program and comparing the performance of two of the programs that used the group of practices with two that did not. The performance measure was derived by summing the number of lines of code and the number of pages of documentation, multiplying them by an estimated program complexity factor, and dividing the result by a person-months figure. The results showed that the programs that used the groups of practices needed only 84-87% of the resources required by programs that did not employ the practices. An interesting note is that the programs averaged 23 lines of code per page of documentation.

The group of practices was referred to as top-down program development and consisted of formal requirements baselining, top-down design including the use of design tools, the holding of formal design reviews, and incremental implementation and associated configuration management. The top-down design tools included a program to produce drawings of the relationship between various parts of the design along with summary tables linking modules to paragraphs in the design document. A form of program design language, referred to as "structured narrative" was used also. The

incremental implementation was done as a series of four builds. These builds were specified as part of the design process, and the stubs were constructed to utilize the same core and processing resources as were estimated for the final project. The opinion of the report's authors seemed to be that the incremental implementation was a particularly desirable practice.

In addition to the design display program cited above, another software tool employed by the project as a MPP was a design flow trace program. This program took as input a specification of the data used by each module and produced a trace of information flow throughout the system. The authors felt, however, that this tool was of reduced usefulness since the information on each module had to be entered manually.

The report also lists a number of software tools that do not, in fact, qualify as MPP. For example, utility programs for saving and restoring drum files from magnetic tape seem to simply make more machine capabilities available to the user, rather than contributing to any of the goals of MPP. Nevertheless, the authors of the report felt that the use of these tools contributed to the enhancement of productivity.

3.1.4 The Boeing Computer Services Experience. The Boeing Computer Services (BCS) report (Black, Katz, Gray, and Curnow, 1977) is based on data collected on five, in-house programming projects: namely, an engineering design support system, a compiler for a cost estimation modeling language, a database management system, a computer system performance analysis program, and a wiring planning system. In comparison to the systems reported on from the other contractors, these were quite small, ranging from 0.6 person-years and 2,890 lines of code to 82.6 person-years and 240,000 lines of code. The data that were collected on each program consisted of person-month figures for each phase of development and of the results of a questionnaire answered by the project manager

questionnaire answered by the project manager about the programming practices used on the project. Analysis of this data involved a formula for estimating the man-months of effort required on a project. This formula, which has been developed on projects that did not use MPP, was based on the amount of kind of programming required in a project and was accurate to 15%.

Using this formula, an effort estimate was made for each of the five projects. Three of the projects were more than 15% below estimated effort, two of them by more than 80%. The questionnaire results were then analyzed to create a list of MPP unique to those projects that were below cost estimates. This list contained the following thirteen practices: formal task assignments; formal reviews, document pertinence; unit development folders; construction plan; review prior to coding; interface conventions; code organization and comments; code verification; review for quality; end item inspection; test formalism, and change control board. Formal task assignments refers to the practice of giving project staff formal, written task assignments. Formal reviews and review prior to coding cover practices related to requirements and design reviews. Document pertinence and unit development folders both refer to particular approaches to controlling and maintaining documentation throughout all phases of a project. The use of a construction plan and a change control board is part of stepwise implementation, though the control boards in this case were apparently occasionally used to resolve inadequacies in the requirements specification. Interface conventions and code organization and comments refer to naming, commentary, and modularity conventions. (All five projects used structured programming constructs.) Finally, code verification, review for quality, end item inspection and test formalism refer to code testing practices; they all require as a prerequisite a sufficiently formal design so that specific, testable functions are assigned to each module of code.

In addition to those practices whose benefits could be empirically validated, the authors felt that two other practices were important in project success as prerequisites for other practices. These were the use of top-down design and design verification. On the other hand, despite their presence in the list, the authors did not feel that interface conventions and coding conventions were significant sources of benefit since their use was mandated by considerations other than contribution to project performance.

3.1.5 The Computer Science Corporation Experience. For their study of the impact of MPP, the Computer Science Corporation (CSC) researchers chose three naval shipboard command and control systems. All of the projects used the CMS-2 language, and the final systems ranged in size from 94,000 words to 310,000 words. No personnel staffing figures are available from the report. In order to evaluate the impact of MPP on these projects, the report authors initially tried to collect resource utilization data that could be used for evaluation. Because of the post-hoc nature of the study, such data proved unavailable. Faced with this difficulty, they then based their analysis on their own expert judgment and on the judgment of other CSC staff.

They began their analysis by identifying twelve different kinds of problems or errors that can occur in a piece of software; examples include incompatibility in data flow between routines, incorrect data manipulation within a routine, and failure to supply a capability specified in a requirements document. For each phase of software development, the percentage of problems or errors falling into each class was estimated. Another matrix was then constructed in which each of 26 MPP was given a value representing the extent to which it reduced each of the twelve types of problems. The product of these two matrices yielded a single overall goodness value for each MPP.

While this procedure is probably superior to just ranking each of the MPP, it does have an unfortunate drawback; use of unweighted error rates does not capture the amount of damage done by an error. As an example, while errors in meeting requirements are common in the analysis phase, they do little damage there; on the other hand, their occurrence in final system integration is often catastrophic. Thus, the conclusion that programmer code reviews have the highest impact of any MPP is suspect. While such reviews do catch many of the types of errors listed, they probably do not catch the most damaging errors, such as failure to meet requirements.

The high ratings given to chief programmer teams and to the practice of having a single individual in charge of each build, on the other hand, seem to have a more solid foundation. Both these practices involve putting important areas of technical control in the hands of a single individual. Likewise, the high rating given to the use of structured walkthroughs also deserves attention. This type of review and the use of design reviews in general also received favorable comment elsewhere in the report. Also noteworthy was the moderate rating given to structured programming, at least in comparison with other techniques.

3.1.6 The Martin Marietta Experience. The Viking project (Prentiss, 1977) conducted by the Martin Marietta Corporation employed three major software systems. The missions operations software consisted of the ground-based programming for mission control and data analysis. It comprised some 280,000 source statements and had approximately 24,000 pages of documentation; these were produced with 148.5 person-years of effort. The flight software consisted of programs for the on-board computers of the Viking orbiter and Viking lander; basic software consisted of 20,000 instructions developed at a cost of 134.0 person-years. An additional 200,000 instructions and 41.1 person-years were invested in emulation,

simulation, and diagnostic support software. The system test software was used to control and monitor hardware tests on the orbiter and lander. It needed 133,000 assembly language statements requiring a 50.0 person-year investment. Since no formal data collection techniques were employed to evaluate this effort, the report represents the experiences and impressions of its authors.

The Viking report is an extremely valuable source of insight both as to the impact that traditional programming practice can have on MPP and as to the effects that inexperienced management can have on MPP success. All of the Viking software efforts were characterized by exceptionally large amounts of time spent in testing and debugging. In the missions operations software, for example, this accounted for 55% of project effort, over and above coding time. Software delivery was apparently frequently behind schedules and only delays in other aspects of the project saved software delivery from being a major roadblock to missions execution.

A variety of sources of difficulty were responsible for these delays. In the case of the missions operations software, at least one of these difficulties was simply inadequacy of computing resources to support software development. First, software development was done at two different sites using four different computing systems from three different manufacturers. While all of these systems supported FORTRAN, the dialects were sufficiently different to require a standardization and conversion effort that added about ten percent (15 person-years) to project requirements. Furthermore, time availability on the target system consisted only of large blocks of time provided in the evenings and on weekends, and the effect of this situation was that testing was often rushed and unsystematic.

A related problem occurred in the development of the system test software. There, the problem was not availability of computing resources, but rather the quality of the resources. The computer, purchased to support the system, was poorly chosen. To quote, "the I/O portion of the computer had design problems, the FORTRAN compiler contained so many errors as to make it useless, the bit/byte instructions worked so slowly that only limited usage was allowable..." (Prentiss, 1977.)

In addition to the computing resources problem, Viking also suffered from problems caused by management with little software expertise. At one point, for example, management arbitrarily ordered a five month speedup in delivery of a part of the system. The result was that other parts of the system, needed to use the programs in question, were not available and the delivered programs could not be used. On another occasion, management ordered delivery of programs prior to integration testing, with the result that many redeliveries of the programs were needed.

Along with these lessons on the effects of inadequate computing resources and poor management, the Viking experience offered some insight into the beneficial effects of good software tools. Two examples are particularly noteworthy. One was a file backup and retrieval system that largely compensated for inadequate disk storage on one of the systems. The other was an emulation system for the on-board computers that allowed substantial software to be developed prior to the final selection of the on-board system; the software sizing obtained from this preparatory work played an important role in this final selection.

In regard to the use of MPP, two practices seemed to be of particular benefit. The use of preparatory builds prior to the delivery of the final software was regarded as essential to the successful development of the missions operations system. Also of value was the use of a management

structure which involved a "cognizant engineer" and a "cognizant programmer." The cognizant engineer was responsible for writing the requirements document for each program and for certifying and accepting the delivered software. The cognizant programmer played much the same role as a chief programmer in a chief programmer team.

3.2 Major Findings

As can be seen from these descriptions, the reports covered quite a wide variety of kinds and environments for software development. With this diversity of situations naturally came a diversity of software practices so that perhaps less than a quarter of the practices used wee common to all the projects. Moreover, it was often difficult to separate the impacts of different techniques within the same project. Any conclusions made from the project experiences must, therefore, necessarily be more impressionistic than analytical. Nevertheless, the studies yielded a few clear conclusions of the relative merits of different MPPs.

3.2.1 Need for Early, Systematic Testing. Practices that lead to systematic, rigorous testing of systems as early as possible in the development period clearly had a beneficial effect in many of the projects. These practices most frequently were incremental implementations in which each build was intended to meet an explicit set of performance criteria. Other frequently used practices were programmer peer code reviews and the use of independent test organizations to verify module accuracy. Two explanations can be suggested for this effect. One is that earlier visibility of deficiencies actually reduces the effort spent on correcting errors, since the effects of the errors do not propagate as far. An alternative explanation is that early error detection does not actually reduce the time spent on error correction, but, because the errors are found sooner, there

is more opportunity to correct errors without schedule slippage. While the first hypothesis has more a priori appeal, none of the studies contained sufficient evidence to refute the second one.

3.2.2 Importance of Firm Specifications. A necessary prerequisite to the success of a MPP (or a conventional programming practice) is the availability of a firm set of specifications as to what tasks the final software product is to perform. While the availability of such a set of specifications is usually automatically assumed, the rigor and clarity of the specifications can have an extreme impact. Several of the projects noted the impact of having, or not having, a baseline of functions that the system was to perform, and one project (performed by TRW) noted changes in specifications as the single greatest problem in system construction.

3.2.3 Formal Designs and Design Reviews. The completion of designs prior to coding and the use of design reviews (also called walkthroughs or structured walkthroughs) were practices that many of the projects found beneficial. While the technique of top-down design received some favorable comment, the reports seemed to indicate that any other techniques that achieved a completely specified design would have an equivalent effect. Design tools, such as HIPO charts and program design languages, were also effective.

3.2.4 A Few Good People. Three of the projects found benefit from management structures that placed substantial technical responsibility in the hands of a few key individuals, whether they were referred to as chief programmer, build leader, or cognizant programmer. The benefit from this organization may occur either because it takes advantage of the technical knowhow and performance of superior individuals or because the individuals, regardless of their own skills, are able to coordinate the activities of others.

3.2.5 Importance of Conventional Practices. As illustrated by the Viking project, in particular, adequate computing resources do not guarantee a project's success, but the lack of them can potentially cause its failure. If computing time is difficult to obtain or if the system being used does not have sufficient memory or file storage, then the positive benefits of MPP are not likely to be realized.

A similar comment holds for the adequacy of support software. One project reported major difficulties due to an inadequate file backup-restore system, while another reported a batch runstream controller as being of significant benefit to the project. MPP are not likely to be of much value in a project unless the project also has available accurate compilers, complete operating systems, and functional support utilities.

The other side of this issue is that abundant machine resources and good support software cannot compensate for poor management or inadequate system design. No matter how available machine time is or how easy it is to manage files, the project will fail unless the programmers are working on the right tasks. Additionally, there was some suggestion that there was a point of diminishing returns for additional machine resources or support software.

3.2.6 Role of Programming Language. Surprisingly, the report contents offered very little comment--positive or negative--on the relative merits of different programming languages. Additionally, the results on structured programming (in the sense of goto-free control structures) from those projects that used it indicate that its effects, if any, are weak. On the other hand, program design languages and standards for commentary received positive evaluations in several projects.

A possible explanation for these findings is that a program can be considered to have two aspects: It can specify (to the machine) the computaion to be performed, and it can communicate to other programmers the original program writer's intent. For projects with many programmers, the later aspect becomes prominant. While both these aspects are usually conveyed primarily by the programming language, this will not be the case if there are other means, such as various kinds of documentation, by which programmers can convey their intent. Relative to the expressive power of documentation such as system designs and module prologues, programming language differences may well be insignificant.

4. CONCLUSIONS AND RESEARCH RECOMMENDATIONS

4.1 General Conclusion

The analysis of the reports provided by the software contractors concerning their experiences with the use of various modern programming practices (MPPs) leads to a general conclusion about the impact of these practices upon software-generation performance. In keeping with the classification of MPP as being either technical, managerial, and/or informational in intention, it seems that the informational domain has the greatest impact. That is, practices classified as being partially or wholly informational were found to be particularly worthwhile and clearly had more impact than those that were solely technical or managerial. Alternately stated, those practices found most advantageous helped programmers learn about how the part of the program written by others worked or was planned to work. As an example, if, because of a failure in the information that has been supplied, a programmer writes a module that does the wrong job, the good technical qualities of the programming language in which it is written or the power of the symbolic debugger used to test it will have little positive effect. Similarly, good management tools for distributing modules among programmers will not be of much use if the module the programmer is told to write does not fit with the rest of the modules.

A decade ago, Nickerson (1969) stated what he felt were significant problems for research in human-computer interaction: namely, (a) development and evaluation of conversational languages; (b) investigation of how use-patterns adopted by users depend on system characteristics, and on systems dynamics in particular; and (c) description or modeling of human-computer interaction. On the basis of the present analysis an additional significant problem might be added: that is, the development and evaluation of tools and techniques for describing how a computer program is constructed and what it does or is expected to do. Such tools and techniques form

the basis for communication among programmers about programs, and the accuracy and completeness of such communication has a profound impact on the system construction process. The availability of, say, a design description device that reduces or eliminates the need for incremental implementation would greatly reduce construction times for most large systems. Research on the evaluation of such description tools and techniques should, therefore, be of high priority for human factors research.

4.2 Some Suggested Research Issues

4.2.1 Techniques for Program Specification. Both the initial specification of a program and the specification of test outcomes involve descriptions of the functions to be performed by the program. The quality of the program, or of the test results, is dependent to a significant extent on the accuracy and completeness of these specifications in reflecting the customer's needs and desires. Human factors research on tools and techniques for extracting and representing this information should, therefore, be useful in improving both the original specification of the design and the MPP used to achieve early, complete testing. As a first example of a kind of research that might be done, consider the problem of describing the relationship between an output and the set of inputs that produce it. One way to do this is by a decision table; an alternate way is by the top levels of a HIPO chart. A worthwhile research activity would be to investigate whether there is a class of problems for which one device is superior to another.

A second example of the kind of research needed comes from the problem of verifying that a set of specifications is complete. One way of doing this is to walk through a simulation of the inputs and outputs of the eventual system. Another is to create a preliminary design for the internal structure of the system and to hold a review of the operation

of this internal design. The human factors research issue here is a determination of the classes of requirements omissions that are best found by each verification technique.

4.2.2 Design Tools. In addition to early testing and completion of specifications, the availability of a complete and rigorous design also was found to have a salubrious effect on project success. An area of research focus that should contribute to the availability of such designs is concerned with tools and techniques for specifying the internal structure of systems. As with research on tools for specification, the questions addressed by research on tools for design should concern the properties of various design tools. An example of such a question is the identification of the relative merits of HIPO charts and program design languages (PDLs) as media for expressing designs. HIPO charts can be seen as focusing information on the flow of data through a system, while PDLs stress the flow of control. Systems that use a wide variety of inputs but do a great deal of processing may be better expressed by a PDL. A research program to investigate this question, or the more general question of the importance of data structures versus control flow, would be of benefit in the selection of design tools.

4.2.3 Interprogrammer Communication. Of all programming practices, those having the greatest success were standards for commenting. An explanation was offered for this superiority in terms of the importance of communicating a program's structure to other programmers via means other than the programming language itself. This suggests investigation of the properties of various techniques for expressing a program's structure. While some work has already been done on the merits of flowcharts, (e.g., Ramsey, Atwood, and Van Doren, 1978; Shneiderman, Mayer, McKay, and Heller, 1977), this line of research needs to be expanded. A useful focus would be on the relationship between the effectiveness of the communication technique used and the characteristics of the program.

4.3 Reorientation of Research Direction

A concluding remark concerns existing human factors research on such topics as control structures for programming languages and commands for interactive systems. As was mentioned earlier, superior conventional practices do not produce superior projects, but inferior resources can cause projects to fail. Much of existing research has been focused on identifying optimum characteristics for programming languages or commands. While such research contributes worthwhile knowledge, a more useful direction to pursue in regard to MPP might be to focus on identifying minimum capabilities that still result in an adequate level of performance to permit project success. A typical study of this type might be aimed at identifying a minimum set of control structure features that would allow effective programs to be written that incorporate the multiplicity of control functions available in conventional languages.

5. REFERENCES

Atwood, M.E., Ramsey, H.R., Hooper, J. N., and Kullas, D.A. Annotated bibliography on human factors in software (ARI Technical Report P-79-1). Alexandria, VA: U.S. Army Research Institute for the Behavioral and Social Sciences, June 1979.

Black, R.K.E., Katz, R., Gray, M.D., and Curnow, R.P. BCS software production data (RADC-TR-77-116). Seattle, WA: Boeing Computer Services, Inc.

Boehm, B. Software and its impact: A quantitative study. Datamation, 1973, 19, 48-59.

Branning, W.E., Willson, D.M., Schaenzer, J.P., and Erickson, W.A. Modern programming practices study report (RADC-TR-77-106). St. Paul, MN: Sperry-Univac Defense Systems, 1977.

Brooks, R. A model of human cognitive processes in writing code for computer programs (Unpublished doctoral dissertation). Pittsburgh, PA: Department of Psychology, Carnegie-Mellon University, 1975.

Brooks, R. Towards a theory of the cognitive processes in computer programming. International Journal of Man-Machine Studies, 1977, 9, 737-742.

Brown, J.R. Impact of MPP on system development. Redondo Beach, CA: TRW Defense and Space Systems Group, 1977.

Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R. Structured programming. London: Academic Press, 1972.

Donahoo, J., Carter, S., Hurt, J., and Farquhar, R. Software production data (RADC-TR-77-177). Huntsville, AL: Computer Sciences Corporation, 1977.

Gould, J.D., and Dronsowski, P. An exploratory study of computer program debugging. Human Factors, 1974, 16, 258-276.

Parnas, D.C. On the criteria to be used in decomposing systems into modules. Communications of the A.C.M., 1972, 15, 1053-1058.

Perry, G., and Willmorth, M.E. An investigation of programming practices in selected Air Force projects (RADC-TR-77-182). Santa Monica, CA: System Development Corporation, 1977.

Prentiss, N.H. Viking software data (RADC-TR-77-168). Denver, CO: Martin Marietta Corporation, 1977.

Ramsey, H.R., Atwood, M.E., and Campbell, G.D. An analysis of software design methodologies (Technical Report 401). Alexandria, VA: U.S. Army Research Institute for the Behavioral and Social Sciences, August 1979.

Ramsey, H.R., Atwood, M.E., and Van Doren, J.R. A comparative study of flowcharts and program design languages for the detailed procedural specification of computer programs (Technical Report TR-78-A22). Alexandria, VA: U.S. Army Research Institute for the Behavioral and Social Sciences, 1978.

Shneiderman, B. Software psychology: Human factors in computer and information systems. Cambridge, MA: Winthrop, 1979.

Shneiderman, B., Mayer, R., McKay, D., and Heller, P. Experimental investigation of the utility of detailed flowcharts in programming. Communications of the A.C.M., 1977, 20, 373-381.

Sime, M.E., Arblaster, A.T., and Green, T.R.G. Reducing programming errors in nested conditionals by prescribing a written procedure. International Journal of Man-Machine Studies, 1977, 9, 119-126.

Sime, M.E., Arblaster, A.T., and Green, T.R.G. Structuring the programmer's task. Journal of Occupational Psychology, 1977, 50, 205-217.

Sime, M.E., Green, T.R.G., and Guest, D.J. Psychological evaluation of two conditional constructs used in computer languages. International Journal of Man-Machine Studies, 1973, 5, 105-113.

Stevens, W.P., Myers, G.J., and Constatine, L.L. Structured design. IBM Systems Journal, 1974, 13, 115-139.

Sussman, G. A computer model of skill acquisition. New York: American Elsevier, 1976.

Weinberg, G.M. The psychology of computer programming. New York: Van Nostrand Reinhold, 1971.

Youngs, E.A. Human errors in programming. International Journal of Man-Machine Studies, 1974, 6, 361-376.